Lecture 20: The Bellman-Ford algorithm and primality testing

We now finish the shortest path problem by studying a different algorithm for the case when edge weights can be negative. In this case, Dijkstra’s algorithm no longer applies (why?). The following two observations are useful. (a) If we apply relaxation on a shortest path from \( s \) to \( u \) in the order of the edges of this path, then these relaxation gives optimal \( d(u) \) value. To get some intuition, consider the example where you have 4 nodes along the path, and each edge weight is 1. Experiment with it to see what happens if you apply relaxation along the path or out-of-order. (b) Relaxation is harmless: if we mix some other relaxation with the key relaxation as in (a), we still will get optimal \( d(u) \). Think about it: why it is correct. These two cases allow us to solve two problems. (i) For a directed acyclic graph. In this case, we first apply topological sorting and then we apply relaxation with the order of edges according to their appearance in the order. The correctness is due to the fact that every path will have edges in one direction. Read the book chapter to make sure you understand this. (ii) For a general graph, we do not know which is the optimal order to apply relaxation. The nice observation by Bellman-Ford algorithm is, if we apply \( n-1 \) rounds of relaxation (while in each round we relax every edge), then we must get shortest path distance to each node. Read the textbook to ensure you understand why. The algorithm runs in \( O(|V||E|) \) time.

Now we switch our attention to a problem arising in number theory. First a little background in number theorem. We define \( x \) modulo \( N \) to be the remainder when \( x \) is divided by \( N \); that is, if \( x = qN + r \) with \( 0 \leq r < N \), then \( x \) modulo \( N \) is equal to \( r \). This gives a notion of equivalence between numbers: \( x \) and \( y \) are congruent modulo \( N \) if they differ by a multiple of \( N \). That is, \( x \equiv y \pmod{N} \) iff \( N \) divides \( x-y \). Modular arithmetic allows usual associative, commutative, and distributive properties of addition and multiplication continue to apply. Although we will not pursue it here, modular addition/multiplication can be done relatively fast.

Now a fact: if \( p \) is a prime number, and \( a \times b \) is divided by \( p \). Then either \( a \) or \( b \) is divided by \( p \) (why?). The primality testing problem is testing whether a given number \( N \) is a prime.

Our main tool is the Fermat’s little theorem: if \( p \) is a prime, then for every \( 1 \leq a < p \), \( a^{p-1} \equiv 1 \pmod{p} \). Here is a simple proof. We first claim: \( a \times i \not\equiv a \times j \pmod{p} \), for all \( 1 \leq i < j \leq p-1 \). For contradiction, suppose \( a \times i \equiv a \times j \pmod{p} \) for some \( i \) and \( j \). Then \( a \times (i-j) \equiv 0 \pmod{p} \). Since \( a < p \) and \( 0 < i-j < p \), this is not possible. So the claim holds.

Thus, \( 1 \times a, 2 \times a, 3 \times a ... (p-1) \times a \) are all distinct modulo \( p \) (and not 0). That is they are some permutation of \( 1, 2, ..., p-1 \). So we multiple them: \( (p-1)!a^{p-1} \equiv (p-1)! \pmod{p} \). Since \( (p-1)! \) is not divided by \( p \), then \( a^{p-1} \equiv 1 \pmod{p} \).

It turns out that for most \( N \), there is a good chance (actually over 50%) of picking one \( a \) s.t. \( a^{N-1} \not\equiv 1 \pmod{N} \) if \( N \) is not a prime. We will prove this claim next time. Thus, if we pick some random number \( a \) between 1 and \( N-1 \), we have 1/2 chance that this a will fail the Fermat’s test if \( N \) is a composite. This implies that (1) when \( N \) is a prime, the randomized algorithm always output correct answer. (2) when \( N \) is a composite, at least 1/2 chance we will know it. We can repeat this procedure \( k \) times. And if one of the \( k \) tests fails the Fermat’s test, we know \( N \) is a composite. If all \( k \) tests pass, we output \( N \) is a composite. The chance of an incorrect answer is at most \( (1/2)^k \), which is very small when \( k \) is large (say 100).